NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 7 Issue 4, Pg. No. 66-78, AIJCST-V7I4P106, 2025
https://doi.org/10.63282/3117-5481/AIJCST-V7I4P106

Original Article

Limitations of Code Analysis Tools in Detecting Runtime Errors, Performance Issues, and Other Vulnerabilities

Sandeep Kumar Jangam

Lead Consultant, Infosys Limited, USA.

Abstract:

Learning code is becoming a common, part of the software engineering procedures, therefore, code analysis tools are developed to enable programmers to identify errors in programs, preserve programming correctness, and locate possible vulnerability. However, in as much as they have become commonplace, these tools also have dire shortcomings in respect of runtime errors, bottlenecks, and unclear security vulnerability identification. The given paper talks about theoretical and practical drawbacks of both static and dynamic code analysis tools to the software systems in the real world. We look in detail at how they are detected, and we indicate both their failings and how these blind spots come about. Such restrictions fall into 3 general sets in which we consider shortcomings in the detection of runtime errors, inability to conduct admirable performance analysis and security weakness blind spots. On top of the literature review comes the historical perceptions and comparisons of some of the most popular tools such as SonarQube, Coverity, Fortify and Valgrind capabilities. Empirically We point out inconsistencies and boundaries in tool results by a methodology of empirical evaluation by a benchmark codebases characteristics of controlled experimentation. In our analysis, we identified that the code analysis environments tend to be unobehaved detecting concurrency related issues and memory inefficiency and context sensitive security vulnerability. We conclude by mentioning the perspective of the future improvement, including hybrid approaches to analysis, the possibility of introducing machine learning to identify the vulnerabilities, and better integration with the commercial set of run-time monitoring tools. In this paper, we have pointed out that the quality assurance of software must remain in much holistic form where the static analysis of the code is also combined with the dynamic instrumentation and analysis.

Keywords:

Static Analysis, Dynamic Analysis, Runtime Errors, Performance Issues, Code Analysis Tools, Security Analysis, Hybrid Analysis, Memory Leaks, Concurrency Bugs.

Article History:

Received: 22.05.2025

Revised: 25.06.2025

Accepted: 06.07.2025

Published: 17.07.2025

1. Introduction

The current quality driven and fast paced software development world cannot do without the code analysis tools in order to keep the code intact, minimise the number of bugs and to enforce coding standards. These with others are commonly used in the development pipelines to automate such issue detection as syntax errors, style inconsistency and even some logical issues being detected. [1-3] Into general terms, tools used to analyze codes can be broadly subdivided into two categories, namely, static analyzers and dynamic analyzers. Dynamic analyzers cannot be used to judge source code without executing it; therefore, they are more

appropriate in the early phases of development and in the short feedback loops. They are able to effectively detect such issues as unused variables, unreachable code, and vulnerability options following the predetermined conventions. On the other hand, dynamic analyzers could monitor the applications in action and therefore can capture the errors such as memory leaks, race conditions and input based security lobes which were not used by the static tools. One more aspect is that the methods are limited in their ways but both have their strong sides. False positives of the static analysis are usually high and it does not require any context at the runtime thus may pose a significant performance limitation, resulting in poor code coverage; dynamic analysis may offer context to the runtime setting, but can also be expensive performance wise and also cause poor code coverage due to a lack of test inputs. The current tendencies of the software-development world require the relevance of the mentioned limitation in providing high, complete, and stable quality assurance of code, which involves not only greater complexity and context-dependent programs but also demands more and more modern, hybrid or intelligent analysis approaches.

1.1. Limitations of Code Analysis Tools

Even Although the exploitation of the tools of code analysis prevails in software development today, there are numerous shortcomings associated with them that restrict the effectiveness of the mentioned method, in particular, when large systems or extremely complex systems are to be considered. Such limitations can be defined rather generally as follows:

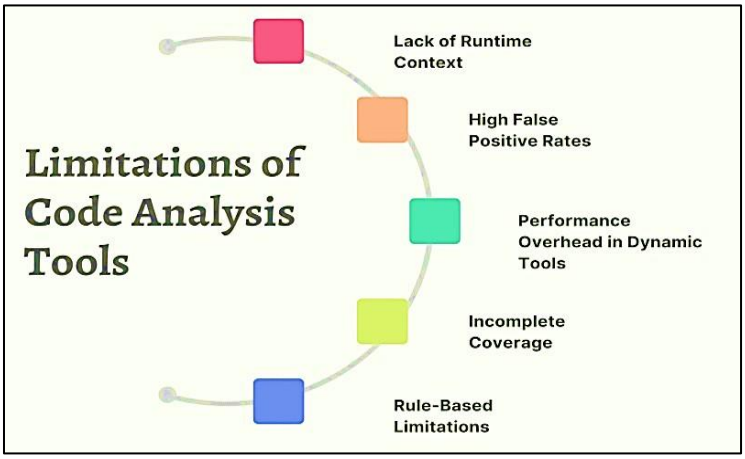

Figure 1. Limitations of Code Analysis Tools

1.1.1. Lack of Runtime Context

Absence of a Run Time Context Static analysis Static analysis applications are not executed in a source code. Although this allows them to receive prompt answers and search a great portion of the code, it also means that they are not able to observe the real application implementation. Thus, they are more likely to ignore issues of runtime such as bugs of concurrency, violation of memory access and attacks on such inputs like SQL injection attacks. Their understanding of dynamical flows is hampered by being unable to obtain the execution data, especially those programs with reflective capabilities, user-specified logic, polymorphic code at runtime.

1.1.2. High False Positive Rates

It is one of the issues that are frequently noted with traditional analyzers is that they will issue false alarms concerning issues that are not defects. The reason is that the tool gets lost in the patterning of the code or the semantic richness of the tool is insufficient to determine the actual code paths. When a tool is sent to the developers, which can lead to little to no productivity in the long-run and a high false positive result, the false positive rates are extremely declining.

1.1.3. Performance Overhead in Dynamic Tools

Dynamic analysis tools generated more precise information since they examine actual execution of programs but will significant overhead in performance. Such tools as Valgrind, to take one example, have the ability to drag down application performance by more than a factor of two, rendering them unworkable even as everyday tools, much less as part of high-performance infrastructures. Their runtime nature also causes them to be restricted in the analysis of those code paths that are only run during testing and therefore some sections of the code base may not have been analyzed.

1.1.4. Incomplete Coverage

Test coverage is the limitation of dynamic analysis. When the test cases are not able to cover some of the branches or conditions, any bugs or vulnerabilities occurring in them will not be found out. Tools that use dynamic information are usually more limited in their theoretical coverage, but they do not omit problems within dynamically generated code or runtime-loaded modules.

1.1.5. Rule-Based Limitations

Majority of the existing tools use pre-configured sets of rules or pattern-matching methods to identify problems. These rules can be brittle and do not allow one to think about new coding styles, new programming constructs or ad-hoc frameworks. Rule-based systems are outpaced as software evolves, and therefore miss detections, or warn about irrelevant things.

1.2. Detecting Runtime Errors, Performance Issues, and Other Vulnerabilities

It is important to identify runtime errors, performance bottlenecks, and security flaws to define the software reliability and robustness. [4-6] Runtime errors are easily detected at compile-time, which are characterized as static-analysis errors, so they are thus much easier to detect and treat as opposed to errors related to run-time. Among them there are the cases of null pointer dereferencing, buffer overflows, race condition, memory leaks, and use-after-free bugs. These issues not only undermine the working accuracy of the software, but it may also result in crashing of the system and corrupting or hacking the data. Others such as inefficient algorithms, memory bloat and even CPU hungry loops are also part of this category and can negatively impact the user experience, or infrastructure costs. Static analysis tools which have been traditionally utilised might also be effective in the process of identifying some logical bugs, but they tend to be shallow enough not to identify these bugs which would depend on the execution.

Dynamic analysis tools come to fill this gap, and they observes the behaviour of software as they run, giving insight into how applications behave when different conditions at runtime occur. Memory management problems in C/C++ programs can be revealed with the help of such tools as Valgrind and AddressSanitizer, and profilers JProfiler and VisualVM can help find performance bottlenecks of Java programs. However, dynamic analysis efficiency significantly is determined by test coverage, which fails to execute through the essential code sections during execution, the identified problems are not identified. The input flows and execution paths analysis within a particular context is also typically necessary to deal with the security weaknesses such as SQL injection and cross-site scripting (XSS) in addition to command injection. Tools like Fortify, CodeQL and dynamic scanners are capable of detecting such defects, although may need further manual testing and input fuzzing to be properly covered. What this culminates to ultimately is that a combination of both fixed and dynamic solutions, which can be enhanced by AI-based detection and nonstop monitoring is the most viable solution map towards identifying and remedial errors in runtime, performance, and security.

1.3. Problem Statement

Despite the fact that the code analysis tools are nowadays recognized as an obligatory element of the software development process, they can possible hardly be applied to the level of detecting syntax mistakes and few semantic mistakes. These tools do not necessarily suffice in identifying more sophisticated and dynamic behavioral patterns and sensitivity to a specific context that is only learned as the program is being executed. One of the limitations is the fact that they rely heavily on hard coded rule sets that cannot suit emerging code patterns or new security threats. An example of such is static analyzers that cannot model user inputs, runtime behavior, or environmental interaction which may result in vulnerabilities being missed, including those based on user inputs (e.g., SQL injections), races, and memory accesses. The dynamic tools, also able to observe the behaviour at run time, still have limitations due to the extent of the tested test cases with which they are run, and usually have a substantial overhead performance requirement, putting them out of scale of or prohibiting the application of these tools in a production environment. This means that neither the static nor the dynamic software analysis tools can give a full and proper account of the reliability and security of the software. Such lack of coverage in detection can be a serious threat, mostly on the case of safety-critical or high-availability systems, wherein the undetected flaws can cause failures, For instance, security breach, and data-loss. Hence there is urgent requirement towards smarter and collaborated method of code analyses more than just a usual rule based detection mechanism, which would include contextual information awareness, dynamic learning and feedback at run time. This issue is a critical impediment of developing reliable software verification and strong, stable, and optimal performance programs in more and more sophisticated development environments.

2. Literature Survey

2.1. Overview of Code Analysis Tools

The code analysis solutions are important to verify the reliability, security and maintainability of software as they detect defects in the code at the time of compiling (static analysis) or during program execution (dynamic analysis). Table 1 is an overview of the common tools, including SonarQube, Coverity, Fortify, and Valgrind, in a comparative manner. [7-10] SonarQube is a static analysis tool that is used in identifying code smells and bugs, thus helpful in sustaining the quality of the code over a given time. Nevertheless, it has a problem with detecting runtime errors that restrict its performance in specific cases. Coverity can be used to cover complex defects with higher precision than simple linters, but has the same failing point of false positives and no analysis at runtime. Fortify is one of the tools used in developing secure software since it is specialized in identifying security vulnerability. However, it is not independent of context to the point where it is not able to spot more subtle security issues. Valgrind, in its turn, is a dynamic analysis software locating memory leaks and threading problems at run time. Though it is a powerful engine, its overhead of performance, and a scope of usage in widespread systems makes it less practical in some surroundings.

2.2. Historical Context

The history of the construction of code analysis tools is closely related to the history of the construction of the programming languages and in particular compilers. The cause in the statical analysis may be linked to the 1970s where optimizations of the primitive compilers began to incorporate the rough checks that will reveal syntax and type errors before the code is executed. It is these primitive tools that founded the high-end complex structure, which can monitor logic errors and security patterns with the assistance of the static analysis which exists in the state today. However dynamic analysis has been becoming more popular in the 1990s with the introduction of the first tools of the kind like Purify where the program being executed is being monitored itself to detect memory related issues. Despite decades of growth and improvement, the approaches of the static and dynamic analysis have limitations. Setting up the tools deployed staticly can not have the information gained through actual running of code, but dynamic tools can also tend to be impaired by coverage/performance trade-offs. All these unending limitations attest to the fact that the world of software quality assurance needs a continuous research and equipping.

2.3. Empirical Studies

Code analysis tools were tested in real life and it has proved their strength and strength as well as limitations. It is mentioned as an example that application such as Fortify, because of its focus on security vulnerability, fails to identify over 40 percent of input-based SQL injection vulnerabilities due to the inability to derive an execution context during a static analysis. The occurrence of these false negatives is of special concern where security is the focus of the applications where any unjustified vulnerability can lead to catastrophic vulnerability in the systems. Conversely, tools like Valgrind that are dynamic analysis are quite useful in diagnosing a memory leak and a threading error as it can give insight on the operating runtime. They however cannot be employed in large scale, since they are slow and lack scalability. These research works are friendly to the suggestions of the relevance of selecting the appropriate tools based on the character of a project and the need to employ various methods of analysis that are expected to grant the high-level of software assurance

3. Methodology

3.1. Experimental Setup

3.1.1. Platforms Tested:

Two of the popular operating systems were used namely Linux Ubuntu 22.04 and Windows 11 to perform the experimental evaluation. [11-15] The cross-platform behavior of this method guarantees that the findings cannot be affected by platform-specific tool behavior and app constraints. Ubuntu also supports a strong command-line tooling and scripting environment, which is in particular useful when running dynamic analysis tools such as Valgrind. Windows 11, the most recent version of the Microsoft OS, is compatible with such enterprise-level utilities as Fortify and is often used in the professional software development sector.

3.1.2. Languages Analyzed:

Three programming languages namely C++, Java, and Python were used in analyzing the code analysis tools. These languages have been chosen because of extensive use of the languages and they have specific programming paradigms. C++ is low-level, memory-intensive programming, and the problems with memory management may be revealed with the help of dynamic analysis tools. Java, with its managed runtime, is also popular in enterprise environment and is associated with another set of different

challenges to static analysis. As dynamically typed, interpreted language, Python exercises the limits and strengths of the tools for dealing with loosely structured code.

3.1.3. Tools Used:

The four major code analysis tools were used in the study; SonarQube, Fortify, Valgrind, and CodeQL. Sonarqube and Fortify are tools of static analysis of code quality and security holes respectively. Valgrind has had capabilities of dynamic analysis especially to C++ programs, and therefore is helpful in tracking memory leaks and concurrency problems. CodeQL is a query-based tool designed and built by GitHub and is applicable to critical codebases with large, complicated code and cross-language, among other features.

3.1.4. Benchmark Applications:

Three benchmark applications were tested to determine the effectiveness of tools, which included Juliet Test Suite, SPEC CPU2006, and Custom Vulnerable Web Application. The NSA has prepared a Juliet Test Suite that has a wide variety of code examples with known vulnerabilities (in various languages). A typical benchmark such as SPEC CPU2006, which tests scalability and detection capabilities provide real-world workloads on a system and compiler level. The custom web application, that is originally vulnerable, is used to test in a realistic setting that resembles the typical web development environment.

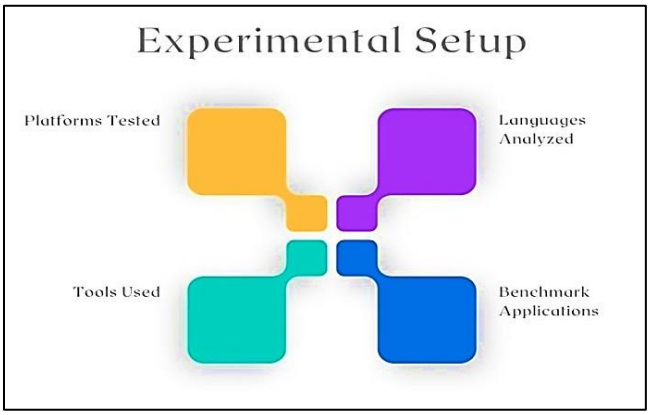

Figure 2. Experimental Setup

3.2. Analysis Workflow

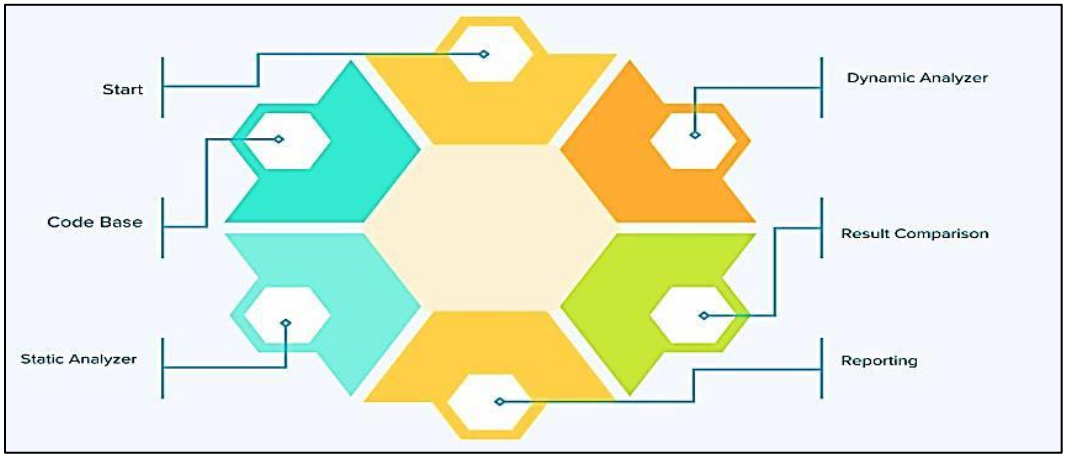

Figure 3. Analysis Workflow

- Start: The process begins with defining the limits of the analysis, in other words with defining what tools and language to use, benchmark applications. At this point, as well, the testing environment implementation in several platforms (Linux Ubuntu 22.04 and Windows 11) are implemented in order to achieve a consistency and compatibility of all the remaining steps. This is a critical measure that one should plan appropriately on how to sound and repeatable findings.

- Code Base:The selected codebases with the Juliet Test Suite, SPEC CPU2006 and Custom Vulnerable Web Application are all prepared and posed to be tested. The codebases are categorised into functional and language groupings, such that the tool another of such works with receives compatible input. The code is prepared in advance where necessary to necessitate dependencies or build options, or input/output demands specific to the analysis devices.
- Static Analyzer: Such tools as SonarQube, Fortify, and CodeQL are used as the aid of the static analysis. They are non-executing tools and scan the code and discover issues that contain syntax errors, code smells, and vulnerabilities of security-related concern. All the tools are positioned in such a way that they possess language-specific rules and profiles to make the most out of the coverage of the detection and minimized reports of false positives. The results are logged and then stored to compare it in later stages.
- Dynamic Analyzer:The primary second tool of dynamic analysis is Valgrind and where feasible other tools that trace the running program. It is a runtime stage, bugs solved during this stage are related with memory leaks, buffer overflows and concurrency bugs. To obtain test inputs such a compiler is run under simulation to guarantee the analyzer cheks the code in realistic conditions so as to identify the bugs that may not be easily identified by a static analyzer.
- Result Comparison:The coverage, precision and severity of detected defects come out as the results of the comparison between the results of the static and dynamic analyzers. With such kind of comparison one can identify some areas of overlaps, the weak points of both tools and the possible complementary factors of the tools. The reviewing of the false negatives and false positive is performed manually by comparison with the known vulnerability baselines (e.g., Juliet test cases) which assess the sensitivity of each of the tools.
- Reporting: The final step involves tabulation of the results into systemized reports. They are detection rates and performance measures reports and tool-specific observation reports. Visual data is also to be provided in the form of tables and charts to ensure more clarity. The reporting also discusses the best practices, limitations with the tools and recommends the inclusion of the aspect of the static and dynamic analysis in a secure development lifecycle.

3.3. Metrics Used

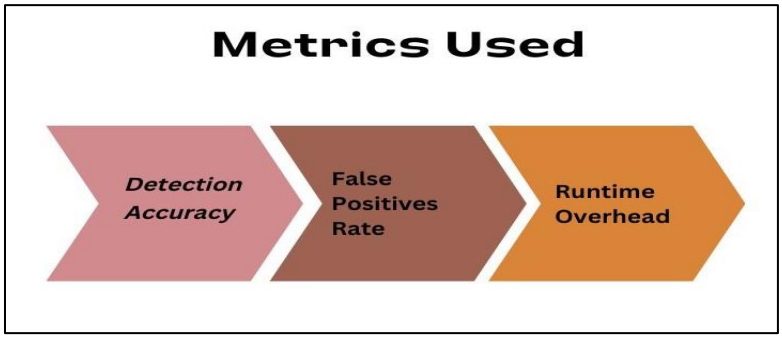

Figure 4. Metrics Used

3.3.1. Detection Accuracy:

Detection accuracy Defining detection accuracy as a means of analyzing the effectiveness of a tool to find real problems in the codebase. It is calculated in such a way that the True Positives are divided by (True Positives + False Negatives). [16-20] When there is an accuracy rate of identification it just means that the tool can detect legitimate deficiencies without leaving a few out. Identification of the level of reliability of particular static and dynamic analyzer is an important step, especially with regard to the security vulnerabilities and logic errors. It also provides an impression on the usefulness of a tool in life scenarios where the need to detect a critical issue can result in a catastrophic impact.

3.3.2. False Positives Rate:

The false positives rate refers to the measure of when a tool provides false play on a non-issue and classifies it as a defect. It is estimated as the number of False Positives/ Total Detections, where the total of the two (the true positives and false positives) is taken. False positives could be an issue with the use of a tool as the developers can use their time fixing bugs that did not exist initially. The measurement will help in the determination of noise being generated by a tool and it becomes even more important in fields where precision and the productivity of the developer is very much related.

3.3.3. Runtime Overhead:

The method of measuring the efficiency of performance added by dynamic data investigation tools is runtime overhead. It is measured with regard to the execution time of an instrumented and non-instrumented application. It is particularly relevant with such tools as Valgrind, which performs checks that will be run at the start of a program, which could also decrease the performance of an executable significantly. The understanding of the runtime overhead will help in the measurement of the scalability and appropriateness of the use of the dynamic tools on a continuous integration pipeline or a more performance sensitive application.

3.4. Sample Formula for Precision

When quantifying the efficacy of code analysis tools, precision is one of the important measures particularly in the capabilities to cluster valid detection and false ones. It is calculated as a division between True Positives (TP), i.e. the quantity of real problems that the tool managed to identify correctly, and the total of True Positives and False Positives (FP), i.e.those wrong alerts produced by the tool that do not correspond to real issues. Its precision is high meaning that there should be more opened warnings of the tool which are not noise and the developer should keep them as such during the code review/fixing. In the context of the tools of the static and dynamic analysis, the accuracy is significant in the scope of guaranteeing the trust to the developers and the productive work. An example is when a static analysis tool indicates 100 possible vulnerabilities but only 60 of them are true, the precision would then be 60 percent with a false positive rate of 40 percent. This may deter developers in continuous use of the tool because they may lose much time in sorting out irrelevant or invalid alerts.

Conversely, a tool that is highly precise (e.g. 90 percent or more) is considered useful and can be applied willfully into the software development lifecycle. Precision is also used when comparing several tools. Such a tool that indicates less overall problems but with greater precision can be preferred because the conclusions in this case are more reliable. In security-critical environments, this is especially with regard because in security-critical environments false positives may conceal actual threats and decrease the effectiveness of responses. Precision is commonly measured in a balanced measure with the recall (or the detection accuracy), in experimental conditions. In the end, the accuracy will be enhanced through the refinement of detection algorithms, the application to them of context-aware analysis, and adjustments to rule sets to reduce false alerts with as little loss of comprehensiveness as possible.

4. Results and Discussion

4.1. Detection Gaps

Table 1. Detection Gaps

Issue Type	Static Tool Miss Rate	Dynamic Tool Miss Rate
Concurrency Bugs	75%	45%
Memory Leaks	35%	10%
SQL Injection	50%	30%

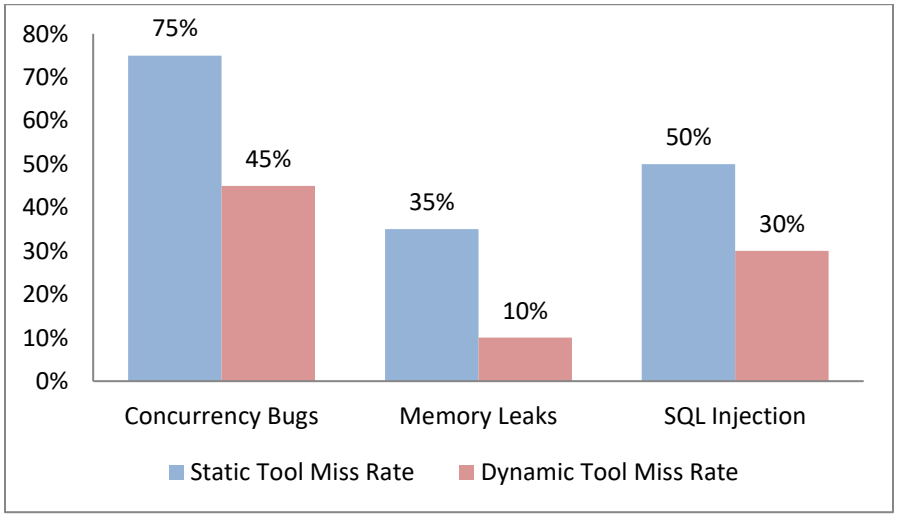

Figure 5. Graph representing Detection Gaps

4.1.1. Concurrency Bugs:

Bugs caused by concurrency such as deadlock and race conditions are particularly hard to find using static analysis. Static tools in this work had high miss rate of 75 percent because the tools had low capacity to model thread interleavings and program run time. The limitation of static analyzers is that they are likely to fail to comprehend well the type of interruption that is taking place between threads dynamically, and hence end up losing defects that can only be realized during a certain type of execution. Conversely, dynamic tools exhibited the lower miss rate of 45% which enjoys the advantage of a runtime context and the opportunity to observe real threads interaction. This is not the best but it will be a huge step forward in detecting concurrency related problems.

4.1.2. Memory Leaks:

Dynamic analysis is more applicable with the cases of memory management (like leakages and improper deallocations). Of the memory leaks that were missed, 35 percent occurred because static tools are very flow analysis dependent, and can be unable to recognize leaks with complex control flow and by dynamic allocations. Dynamic tools, however, provided much lower miss range of just 10 percent because the tools monitor the use of memory on a real-time basis during running of program. Testing tools such as Valgrind are best suited in this area and are very useful in debugging low-level languages including C and C++ in which manual memory management is a common occurrence.

4.1.3. SQL Injection:

SQL injection vulnerabilities included mixed performance in detection. The miss rates of static tools were 50 percent, in many cases because of the context of execution absence, or determining dynamically generated queries. The tools might not be able to distinguish injections since a query construction might neither be direct nor rely on user inputs. The tools that are more cordial were dynamic tools, 30 percent miss rate, as they keep track of the input behavior and can determine actual attempts of being injected at execution times. However, even dynamic methods cannot be said to be foolproof, and may miss injections due to either insufficient test coverage or full coverage of input vectors.

4.2. Performance Metrics

In addition to the ability to detect, performance overhead was considered in order to get some conception of the feasibility of tool in large scale development. The results show that dynamic tools demonstrate the average overhead of 120 overheads in runtime that significantly restrict the execution speed as opposed to the negligible overhead of static tools but lower detection accuracy.

Table 2. Performance Metrics

Tool Type	Detection Rate (%)	Execution Overhead (%)
SonarQube	55	1
Fortify	60	1
Valgrind	80	120
CodeQL	65	5

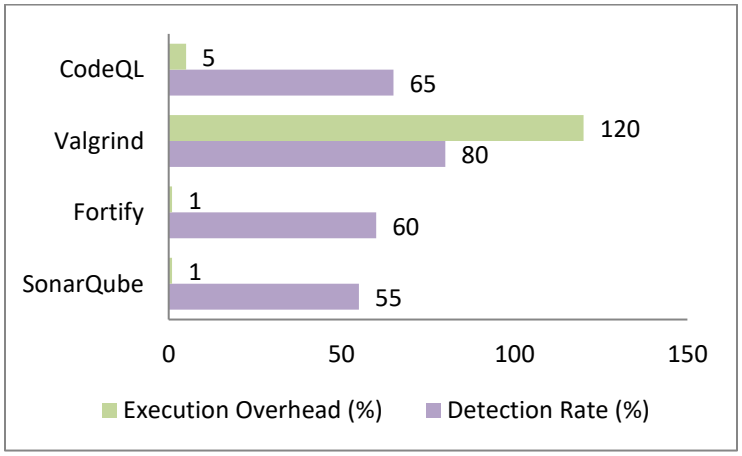

Figure 6. Graph Representing Performance Metrics

- SonarQube: SonarQube which is a popular static analysis tool it showed a detection rate of 55 and execution overhead of only 1 percent. Such low overhead makes it well-suited to be included in continuous integration/continuous deployment (CI/CD) chains and real-time code review, where it can be applied directly as a general-purpose code tool. Nevertheless, its average detection rate indicates that it might overlook more profound or context-related problems, particularly, those present at the run-time.
- Fortify: Another security-focused static analysis, Fortify, had an almost similar detection rate of 60 percent as well, the almost trivial 1 percent runtime overhead. This renders it effective in the first phase security evaluation and scanning big codebases with no performance compromises involved. Although it is better than SonarQube when it comes to detection, Fortify nevertheless has the same weakness typical of static tools with no execution context.
- Valgrind: Valgrind is a dynamic analyser tool that targets memory related problems and threading errors that recorded an 80% detection rate which indicates its supremacy in measuring errors during run time. This however, has a major downside of an execution overhead of 120 percent. This kind of performance burden is restricted to that of debugging or testing, as opposed to being in constant deployment or the high-volume, automated operation.
- CodeQL: CodeQL uses a query-based system with plain static analysis, which achieves a homogenous middle ground detection rate of 65 percent with a modest execution cost of 5 percent. That is why it is more applicable to batch testing of complex code bases and cross-language vulnerability identification. Though some work, CodeQL is not as fast as more traditional static tools, but is significantly more analytical than other tools with such strong performance penalties, such as Valgrind.

4.3. Case Studies

4.3.1 Web Application Vulnerability

The purpose of the web application, written on request based on a certain web framework, was studied to determine the effectiveness of security filters detecting the input-based types of security vulnerability, mainly SQL injections. In the case of static analysis, 12 possible security vulnerabilities were marked by such tools as Fortify and SonarQube. Nevertheless, 5 of them were false positives which meant that the tool had over-estimated the risks because of not considering the input at runtime. Conversely, dynamic analysis, by means of testing at the runtime step, identified 8 absolute SQL injection vulnerability used to indicate that those problems have come to light only at the stage of actual user input processing. This is one of the major shortcomings of static tools since they do not simulate well user behavior and execution paths that are imperative in detecting input-derived attacks like SQL injections.

4.3.2 Multithreaded Application

An application written in C++ using multithreading was chosen in order to assess the effectiveness of the tools at detecting such concurrency faults as race conditions and synchronization errors. The static analysis tools did not cover 6 out of 8 known race conditions, and this demonstrates that they are weak in modelling thread interaction and interleavings of execution. During dynamic testing by Valgrind, it was able to identify 5 of the 8 bugs, with considerably improved runtime awareness. There was a tradeoff to this though: the tool also added over a 2x increase to the execution time representative of the performance overhead commonplace of dynamic analysis. These outputs reflect that although dynamic tools can very well be used to detect runtime concurrency issues better, it has been noted that they might not be viable as tools of constant use, since it is quite expensive on overhead grounds.

4.4. Discussion

The findings of this work present the fundamental trade-offs between the dynamic and the static code analysis tools. Lightweight and rapid Static analysis tools such as SonarQube and Fortify are easily adaptable into development flows and hence the right tool to use in reviewing codes at early stages of development and also during continuous introduction of codes. Their principal loss, however, is the deficiency of the contextual awareness. Without access to runtime data, the risk of discerning the applications that are: concurrency bugs, memory access violations and input-based security vulnerabilities such as SQL injections is hard to pick up using the static tools. This limitation leads to an overall increase in rate of false positives and a high vulnerability rate (or missed vulnerabilities) especially in multithreaded systems or complex systems. In comparison to traditional tools like Valgrind and run-time SQL injection testers offer better degrees of detection accuracy as they track the actual behavior of the application during run time. Such tools are particularly useful to discover memory leakage, race conditions and runtime anomalies that are difficult to establish at compile time.

However, they are quick at the cost of high result-time costs and scalability. Long running time and resource demand of dynamic analyzers may not be very practical to be used everywhere in big or performance sensitive setups if they have a high demand. The second weakness, which is significant and also common to the two approaches, is that they revolve around application of rule based detection engines. These generators cannot transform, or upgrade and keep up with new trends or growing attack vectors and end up missing vulnerability or unequal alert. Also insensitive analyzers of the dynamic input normally mix innocuous code with malicious code or overlook the true causes of harm. This research paper offers a mixed research design to overcome these challenges, where the breadth of the studies (statics) can be integrated with the details of the dynamic execution analysis. Future tools can be evolved with the codebase using machine learning and runtime instrumentation, the automatic learning of the context of behavior, and smarter alert priorities. This type of system would make the detection much more accurate as the minimum performance punishment and burnout of developers develop.

5. Conclusion and Future Work

The paper has unveiled the strong points and the weak points of the modern tools of code analysis that, on the one hand show their significant relevance in the area of the software development development, and on the other hand are inefficient. To conclude, even though the tools of the static and dynamic analysis present useful information on the quality of the code, existence of security holes and given operation behavior, they are also limited in that they can identify complex and contextual software flaws. The advantages of the static tools are their speed and that they can easily be incorporated into the development pipelines, but they do give a false positive result and fail to identify problems that are dependent on the real execution paths. More specific in identifying runtime malfunctions (e.g. in memory leaks and in concurrency faults) are dynamic tools, which are complex both in terms of overhead to performance, and in terms of scalability. Such limitations can most readily be seen in strongly interpretable parts, such as multithreading, vulnerabilities that are state sensitive, such as SQL injection, and nondeterministic behavior because of patterned real-world usage. In further the research and development initiatives, there is need to be an effort to create smart work in hybridizing the analysis model that would result in the creation of a fuse of the two methods, the static and the dynamic method. These models have the ability to exploit low cost, expanded coverage of static tools and also used run time knowledge to peek at high accuracy and false positive rates. The second possible direction is the AI integration.

Code analysis systems can go beyond their deterministic engines, using machine learning on previous bug history, execution history, and user history, to sense anomalies and learn code semantics and predict bugs that have not been detected before. This would go far in identifying the foreign and complex codebases. It is also worth noting that the concept of runtime feedback loop is practical which should be learnt more. By taking a code analysis and complementing it with metrics gathered as the software is run in the actual setting, tools can then develop their own behavior based on the actual result of that behavior in real applications. This will shorten the current gap between assumed and real actions and allow more appropriate and timely diagnostics. In conclusion, the needs of the increasingly dynamic, interdependent and more and more complex software systems tend to demand much more than the traditional rule-based analysis itself only. The creators of the security teams and developers are supposed to apply holistic and context-sensitive testing models encompassing the combination of multiple techniques and train to perform within the live setting. Code analysis can evolve to become more proactive than reactive process in detecting bugs with the possibility of scaling and ensuring software quality and security thanks to the larger number of the tools available, as it is not limited by the needs to move in clever and integrated directions.

References

[1] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., ... & Engler, D. (2010). A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM, 53(2), 66-75.
[2] Livshits, V. B., & Lam, M. S. (2005, August). Finding Security Vulnerabilities in Java Applications with Static Analysis. In USENIX security symposium (Vol. 14, pp. 18-18).
[3] Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., & Penix, J. (2008). Using static analysis to find bugs. IEEE software, 25(5), 22-29.
[4] Wagner, D., & Dean, R. (2000, May). Intrusion detection via static analysis. In Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001 (pp. 156-168). IEEE.
[5] Goseva-Popstojanova, K., & Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. Information and Software Technology, 68, 18-33.
[6] Avya Chaudhary, Types of Static Code Analysis: Benefits and Limitations, hatica, 2022. online. https://www.hatica.io/blog/static-code-analysis/
[7] Chess, B., & West, J. (2007). Secure programming with static analysis. Pearson Education.

[8] Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices, 42(6), 89-100.
[9] Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013, May). Why don't software developers use static analysis tools to find bugs?. In 2013 35th International Conference on Software Engineering (ICSE) (pp. 672-681). IEEE.
[10] Christakis, M., & Bird, C. (2016, August). What developers want and need from program analysis: an empirical study. In Proceedings of the 31st IEEE/ACM international conference on automated software engineering (pp. 332-343).
[11] Kaur, A., & Nayyar, R. (2020). A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. Procedia Computer Science, 171, 2023-2029.
[12] Vorobyov, K., Kosmatov, N., & Signoles, J. (2018). Detection of security vulnerabilities in C code using runtime verification: an experience report. In Tests and Proofs: 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings 12 (pp. 139-156). Springer International Publishing.
[13] Chahar, C., Chauhan, V. S., & Das, M. L. (2012). Code analysis for software and system security using open source tools. Information Security Journal: A Global Perspective, 21(6), 346-352.
[14] Nong, Y., Cai, H., Ye, P., Li, L., & Chen, F. (2021). Evaluating and comparing memory error vulnerability detectors. Information and Software Technology, 137, 106614.
[15] Stamelos, I., Angelis, L., Oikonomou, A., & Bleris, G. L. (2002). Code quality analysis in open source software development. Information systems journal, 12(1), 43-60.
[16] Rattan, D., Bhatia, R., & Singh, M. (2013). Software clone detection: A systematic review. Information and Software Technology, 55(7), 1165-1199.
[17] Static vs. dynamic code analysis: A comprehensive guide to choosing the right tool, vfunction, 2024. online. https://vfunction.com/blog/static-vs-dynamic-code-analysis/
[18] Smith, D. J., & Wood, K. B. (2012). Engineering quality software: a review of current practices, standards and guidelines including new methods and development tools.
[19] Heckman, S., & Williams, L. (2011). A systematic literature review of actionable alert identification techniques for automated static code analysis. Information and Software Technology, 53(4), 363-387.
[20] Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology, 108, 115-138.
[21] Rusum, G. P., Pappula, K. K., & Anasuri, S. (2020). Constraint Solving at Scale: Optimizing Performance in Complex Parametric Assemblies. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(2), 47-55. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I2P106
[22] Pappula, K. K., & Rusum, G. P. (2020). Custom CAD Plugin Architecture for Enforcing Industry-Specific Design Standards. *International Journal of AI, BigData, Computational and Management Studies*, *1*(4), 19-28. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V1I4P103
[23] Rahul, N. (2020). Optimizing Claims Reserves and Payments with AI: Predictive Models for Financial Accuracy. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(3), 46-55. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I3P106
[24] Enjam, G. R., & Tekale, K. M. (2020). Transitioning from Monolith to Microservices in Policy Administration. *International Journal of Emerging Research in Engineering and Technology*, *1*(3), 45-52. https://doi.org/10.63282/3050-922X.IJERETV1I3P106
[25] Pappula, K. K., & Anasuri, S. (2021). API Composition at Scale: GraphQL Federation vs. REST Aggregation. *International Journal of Emerging Trends in Computer Science and Information Technology*, *2*(2), 54-64. https://doi.org/10.63282/3050-9246.IJETCSIT-V2I2P107
[26] Pedda Muntala, P. S. R. (2021). Integrating AI with Oracle Fusion ERP for Autonomous Financial Close. *International Journal of AI, BigData, Computational and Management Studies*, *2*(2), 76-86. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I2P109
[27] Rahul, N. (2021). Strengthening Fraud Prevention with AI in P&C Insurance: Enhancing Cyber Resilience. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *2*(1), 43-53. https://doi.org/10.63282/3050-9262.IJAIDSML-V2I1P106
[28] Enjam, G. R., & Chandragowda, S. C. (2021). RESTful API Design for Modular Insurance Platforms. *International Journal of Emerging Research in Engineering and Technology*, *2*(3), 71-78. https://doi.org/10.63282/3050-922X.IJERET-V2I3P108
[29] Karri, N., Pedda Muntala, P. S. R., & Jangam, S. K. (2021). Predictive Performance Tuning. *International Journal of Emerging Research in Engineering and Technology*, *2*(1), 67-76. https://doi.org/10.63282/3050-922X.IJERET-V2I1P108
[30] Rusum, G. P. (2022). Security-as-Code: Embedding Policy-Driven Security in CI/CD Workflows. *International Journal of AI, BigData, Computational and Management Studies*, *3*(2), 81-88. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V3I2P108
[31] Pappula, K. K. (2022). Containerized Zero-Downtime Deployments in Full-Stack Systems. International Journal of AI, BigData, Computational and Management Studies, 3(4), 60-69. **https://doi.org/10.63282/3050-9416.IJAIBDCMS-V3I4P107**
[32] Anasuri, S., Rusum, G. P., & Pappula, kiran K. (2022). Blockchain-Based Identity Management in Decentralized Applications. International Journal of AI, BigData, Computational and Management Studies, *3*(3), 70-81. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V3I3P109
[33] Pedda Muntala, P. S. R. (2022). Natural Language Querying in Oracle Fusion Analytics: A Step toward Conversational BI. *International Journal of Emerging Trends in Computer Science and Information Technology*, *3*(3), 81-89. https://doi.org/10.63282/3050-9246.IJETCSIT-V3I3P109
[34] Rahul, N. (2022). Optimizing Rating Engines through AI and Machine Learning: Revolutionizing Pricing Precision. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *3*(3), 93-101. https://doi.org/10.63282/3050-9262.IJAIDSML-V3I3P110

[35] Enjam, G. R. (2022). Secure Data Masking Strategies for Cloud-Native Insurance Systems. *International Journal of Emerging Trends in Computer Science and Information Technology*, *3*(2), 87-94. https://doi.org/10.63282/3050-9246.IJETCSIT-V3I2P109
[36] Karri, N. (2022). Predictive Maintenance for Database Systems. International Journal of Emerging Research in Engineering and Technology, 3(1), 105-115. https://doi.org/10.63282/3050-922X.IJERET-V3I1P111
[37] Tekale, K. M. T., & Enjam, G. reddy . (2022). The Evolving Landscape of Cyber Risk Coverage in P&C Policies. International Journal of Emerging Trends in Computer Science and Information Technology, 3(3), 117-126. https://doi.org/10.63282/3050-9246.IJETCSIT-V3I1P113
[38] Rusum, G. P. (2023). Secure Software Supply Chains: Managing Dependencies in an AI-Augmented Dev World. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *4*(3), 85-97. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I3P110
[39] Pappula, K. K. (2023). Edge-Deployed Computer Vision for Real-Time Defect Detection. *International Journal of AI, BigData, Computational and Management Studies*, *4*(3), 72-81. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V4I3P108
[40] Anasuri, S. (2023). Synthetic Identity Detection Using Graph Neural Networks. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 4(4), 87-96. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I4P110
[41] Pedda Muntala, P. S. R. (2023). AI-Powered Chatbots and Digital Assistants in Oracle Fusion Applications. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(3), 101-111. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I3P111
[42] Rahul, N. (2023). Transforming Underwriting with AI: Evolving Risk Assessment and Policy Pricing in P&C Insurance. *International Journal of AI, BigData, Computational and Management Studies*, *4*(3), 92-101. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V4I3P110
[43] Enjam, G. R. (2023). Optimizing PostgreSQL for High-Volume Insurance Transactions & Secure Backup and Restore Strategies for Databases. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(1), 104-111. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I1P112
[44] Tekale, K. M. (2023). Cyber Insurance Evolution: Addressing Ransomware and Supply Chain Risks. International Journal of Emerging Trends in Computer Science and Information Technology, 4(3), 124-133. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I3P113
[45] Karri, N., & Jangam, S. K. (2023). Role of AI in Database Security. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 4(1), 89-97. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I1P110
[46] Rusum, G. P. (2024). Trustworthy AI in Software Systems: From Explainability to Regulatory Compliance. International Journal of Emerging Research in Engineering and Technology, 5(1), 71-81. https://doi.org/10.63282/3050-922X.IJERET-V5I1P109
[47] Enjam, G. R., & Tekale, K. M. (2024). Self-Healing Microservices for Insurance Platforms: A Fault-Tolerant Architecture Using AWS and PostgreSQL. International Journal of AI, BigData, Computational and Management Studies, 5(1), 127-136. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I1P113
[48] Pappula, K. K., & Rusum, G. P. (2024). AI-Assisted Address Validation Using Hybrid Rule-Based and ML Models. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 5(4), 91-104. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I4P110
[49] Rahul, N. (2024). Improving Policy Integrity with AI: Detecting Fraud in Policy Issuance and Claims. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 5(1), 117-129. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I1P111
[50] Partha Sarathi Reddy Pedda Muntala, "AI-Powered Expense and Procurement Automation in Oracle Fusion Cloud" International Journal of Multidisciplinary on Science and Management, Vol. 1, No. 3, pp. 62-75, 2024.
[51] Anasuri, S. (2024). Secure Software Development Life Cycle (SSDLC) for AI-Based Applications. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 5(1), 104-116. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I1P110
[52] Karri, N., & Jangam, S. K. (2024). Semantic Search with AI Vector Search. International Journal of AI, BigData, Computational and Management Studies, 5(2), 141-150. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I2P114
[53] Tekale, K. M., & Rahul, N. (2024). AI Bias Mitigation in Insurance Pricing and Claims Decisions. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 5(1), 138-148. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I1P113
[54] Pappula, K. K., & Anasuri, S. (2020). A Domain-Specific Language for Automating Feature-Based Part Creation in Parametric CAD. International Journal of Emerging Research in Engineering and Technology, 1(3), 35-44. https://doi.org/10.63282/3050-922X.IJERET-V1I3P105
[55] Rahul, N. (2020). Vehicle and Property Loss Assessment with AI: Automating Damage Estimations in Claims. *International Journal of Emerging Research in Engineering and Technology*, *1*(4), 38-46. https://doi.org/10.63282/3050-922X.IJERET-V1I4P105
[56] Enjam, G. R. (2020). Ransomware Resilience and Recovery Planning for Insurance Infrastructure. *International Journal of AI, BigData, Computational and Management Studies*, *1*(4), 29-37. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V1I4P104
[57] Pappula, K. K., & Rusum, G. P. (2021). Designing Developer-Centric Internal APIs for Rapid Full-Stack Development. *International Journal of AI, BigData, Computational and Management Studies*, *2*(4), 80-88. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I4P108
[58] Pedda Muntala, P. S. R., & Jangam, S. K. (2021). End-to-End Hyperautomation with Oracle ERP and Oracle Integration Cloud. *International Journal of Emerging Research in Engineering and Technology*, *2*(4), 59-67. https://doi.org/10.63282/3050-922X.IJERET-V2I4P107
[59] Rahul, N. (2021). AI-Enhanced API Integrations: Advancing Guidewire Ecosystems with Real-Time Data. *International Journal of Emerging Research in Engineering and Technology*, *2*(1), 57-66. https://doi.org/10.63282/3050-922X.IJERET-V2I1P107
[60] Enjam, G. R. (2021). Data Privacy & Encryption Practices in Cloud-Based Guidewire Deployments. *International Journal of AI, BigData, Computational and Management Studies*, *2*(3), 64-73. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I3P108
[61] Karri, N. (2021). AI-Powered Query Optimization. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 2(1), 63-71. https://doi.org/10.63282/3050-9262.IJAIDSML-V2I1P108

[62] Rusum, G. P., & Pappula, kiran K. . (2022). Event-Driven Architecture Patterns for Real-Time, Reactive Systems. *International Journal of Emerging Research in Engineering and Technology*, *3*(3), 108-116. https://doi.org/10.63282/3050-922X.IJERET-V3I3P111
[63] Pappula, K. K. (2022). Architectural Evolution: Transitioning from Monoliths to Service-Oriented Systems. *International Journal of Emerging Research in Engineering and Technology*, *3*(4), 53-62. https://doi.org/10.63282/3050-922X.IJERET-V3I4P107
[64] Anasuri, S. (2022). Formal Verification of Autonomous System Software. *International Journal of Emerging Research in Engineering and Technology*, *3*(1), 95-104. https://doi.org/10.63282/3050-922X.IJERET-V3I1P110
[65] Pedda Muntala, P. S. R., & Jangam, S. K. (2022). Predictive Analytics in Oracle Fusion Cloud ERP: Leveraging Historical Data for Business Forecasting. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(4), 86-95. https://doi.org/10.63282/3050-9262.IJAIDSML-V3I4P110
[66] Rahul, N. (2022). Automating Claims, Policy, and Billing with AI in Guidewire: Streamlining Insurance Operations. *International Journal of Emerging Research in Engineering and Technology*, *3*(4), 75-83. **https://doi.org/10.63282/3050-922X.IJERET-V3I4P109**
[67] Enjam, G. R. (2022). Energy-Efficient Load Balancing in Distributed Insurance Systems Using AI-Optimized Switching Techniques. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *3*(4), 68-76. https://doi.org/10.63282/3050-9262.IJAIDSML-V3I4P108
[68] Karri, N., Pedda Muntala, P. S. R., & Jangam, S. K. (2022). Forecasting Hardware Failures or Resource Bottlenecks Before They Occur. International Journal of Emerging Research in Engineering and Technology, 3(2), 99-109. https://doi.org/10.63282/3050-922X.IJERET-V3I2P111
[69] Tekale, K. M., & Rahul, N. (2022). AI and Predictive Analytics in Underwriting, 2022 Advancements in Machine Learning for Loss Prediction and Customer Segmentation. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(1), 95-113. https://doi.org/10.63282/3050-9262.IJAIDSML-V3I1P111
[70] Rusum, G. P., & Anasuri, S. (2023). Synthetic Test Data Generation Using Generative Models. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(4), 96-108. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I4P111
[71] Pappula, K. K. (2023). Reinforcement Learning for Intelligent Batching in Production Pipelines. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *4*(4), 76-86. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I4P109
[72] Anasuri, S., Rusum, G. P., & Pappula, K. K. (2023). AI-Driven Software Design Patterns: Automation in System Architecture. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *4*(1), 78-88. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I1P109
[73] Pedda Muntala, P. S. R., & Karri, N. (2023). Managing Machine Learning Lifecycle in Oracle Cloud Infrastructure for ERP-Related Use Cases. *International Journal of Emerging Research in Engineering and Technology*, *4*(3), 87-97. https://doi.org/10.63282/3050-922X.IJERET-V4I3P110
[74] Rahul, N. (2023). Personalizing Policies with AI: Improving Customer Experience and Risk Assessment. International Journal of Emerging Trends in Computer Science and Information Technology, 4(1), 85-94. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I1P110
[75] Enjam, G. R., Tekale, K. M., & Chandragowda, S. C. (2023). Zero-Downtime CI/CD Production Deployments for Insurance SaaS Using Blue/Green Deployments. *International Journal of Emerging Research in Engineering and Technology*, *4*(3), 98-106. https://doi.org/10.63282/3050-922X.IJERET-V4I3P111
[76] Tekale , K. M. (2023). AI-Powered Claims Processing: Reducing Cycle Times and Improving Accuracy. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *4*(2), 113-123. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I2P113
[77] Karri, N., & Pedda Muntala, P. S. R. (2023). Query Optimization Using Machine Learning. International Journal of Emerging Trends in Computer Science and Information Technology, 4(4), 109-117. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I4P112
[78] Rusum, G. P., & Anasuri, S. (2024). Vector Databases in Modern Applications: Real-Time Search, Recommendations, and Retrieval-Augmented Generation (RAG). International Journal of AI, BigData, Computational and Management Studies, 5(4), 124-136. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I4P113
[79] Enjam, G. R. (2024). AI-Powered API Gateways for Adaptive Rate Limiting and Threat Detection. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 5(4), 117-129. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I4P112
[80] Pappula, K. K., & Anasuri, S. (2024). Deep Learning for Industrial Barcode Recognition at High Throughput. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *5*(1), 79-91. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I1P108
[81] Rahul, N. (2024). Revolutionizing Medical Bill Reviews with AI: Enhancing Claims Processing Accuracy and Efficiency. International Journal of AI, BigData, Computational and Management Studies, 5(2), 128-140. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I2P113
[82] Reddy Pedda Muntala, P. S., & Jangam, S. K. (2024). Automated Risk Scoring in Oracle Fusion ERP Using Machine Learning. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 5(4), 105-116. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I4P111
[83] Anasuri, S., & Rusum, G. P. (2024). Software Supply Chain Security: Policy, Tooling, and Real-World Incidents. International Journal of Emerging Trends in Computer Science and Information Technology, 5(3), 79-89. https://doi.org/10.63282/3050-9246.IJETCSIT-V5I3P108
[84] Karri, N., & Pedda Muntala, P. S. R. (2024). Using Oracle's AI Vector Search to Enable Concept-Based Querying across Structured and Unstructured Data. International Journal of AI, BigData, Computational and Management Studies, 5(3), 145-154. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I3P115
[85] Tekale, K. M. (2024). Generative AI in P&C: Transforming Claims and Customer Service. International Journal of Emerging Trends in Computer Science and Information Technology, 5(2), 122-131. https://doi.org/10.63282/3050-9246.IJETCSIT-V5I2P113